```
;; -*- mode: asm; mode: outline-minor-mode; outline-regexp: ";;;+" -*-

;;; PIC16LF819

        ;; boot.asm      version 20060722
        ;; copyright 2006 Frank Sergeant (frank@pygmy.utoh.org)
        ;; MIT/BSD style license,
        ;; (http:/pygmy.utoh.org if you wish to see the exact license)

        ;; This version uses the 1-way serial port, from PC's TX pin
        ;;   into PIC's RA5/Vpp/MCLR* pin.  See also boot2.asm which
        ;;   uses a 2-way serial port (thus with feedback).

        ;; Here is the gpasm command to assemble this file.  Change the -I
        ;;  parameter to suit your situation
        ;;
        ;;      gpasm -I../include boot.asm
        ;;
        ;;  one way to see a dump of the contents of the resulting
        ;;  hex file is to run
        ;;
        ;;      gpdasm -i -p 16f819 dummy1.hex
        ;;

        ;; 16F877 allows writes to program memory in blocks of 4 words, with automatic
        ;;  block erasure.  (Approximately 4 ms stall per block.)

        ;; 16F819 requires explicit 32-word block erase (and then 4-word block writes)
        ;;  (approximately 2 ms stall for the block erase and 2 ms for the 4-word write).

        ;; So, on the '819, this affects either where user program
        ;; (and user interrupt handler) can reside or it requires user
        ;; program to be ORG'd at 0x0000 and user code to include the
        ;; jump to the bootloader.

        ;; The host PC downloader uses a single line from the PC serial
        ;; port TX (DB9 pin 3, through potentiometer to scale voltage
        ;; between -5V and 5V) into PIC pin 4 (RA5/Vpp/MCLR*), plus a
        ;; ground connection (from PC serial port (DB9 pin 5)).

        ;; So, this is a 2-line serial connection (transmit from the PC
        ;; plus ground).  The host PC downloader program transmits the
        ;; application blind, with no feedback.  This costs us only a
        ;; single pin on the PIC, which is the only pin that cannot be an
        ;; output pin.  Of course, you can use any pin you wish for this
        ;; (but adjust RXPIN, RXPORT, RXTRIS equates accordingly).

        ;; This particular pin (RA5/Vpp/MCLR*) was chosen because it is the only
        ;; pin that is input only.  This saves the other pins (which can be either
        ;; inputs or outputs) for the application.  The bootloader sets
        ;; the configuration word so that RA5/Vpp/MCLR* is set as a digital pin
        ;; rather than an MCLR* pin.  A high-voltage (nominally 13 V) on that pin
        ;; will still put the PIC into high-voltage programming mode.  So, we scale
        ;; the voltage from the PC serial port through a voltage divider (I use a
        ;; small 10K or 20K potentiometer) so that the voltage does not go higher
        ;; than about 5 V nor lower than about -5 V).

        ;; See the file pinouts.txt for an example of the serial connections.

        ;; If your application has a spare I/O pin and if you would rather use a
        ;; different pin (perhaps one with diode clamps so that not even the
        ;; potentiometer is needed).  All you need to do is change three equ
        ;; statements in boot.asm.  (See RXPIN, RXPORT, and RXTRIS.)

        ;; The bootloader code uses the first block (0x0000-0x001F) for jumping
        ;; to the bootloader code proper and for the interrupt vector (which jumps
        ;; to the user's interrupt vector at 0x0020).  It uses high memory for
```

```
        ;; the bootloader code proper.  Be sure to change the org statement
        ;; to suit the version of the PIC chip you are using.  For the 16F819,
        ;; the bootloader code is org'd at around 0x0749.  For the 16F818 (with only
        ;; one K words of memory), it should be org'd at around 0x0349.

        ;; If you change where the bootloader is org'd, a corresponding change should
        ;; be made to the host PC downloader program (e.g. picbl.py) so it will check
        ;; the length of the application code so the bootloader code in the PIC will
        ;; not be overwritten.

        ;; Note, on the 16F818 and 16F819 erasures must occur in blocks of 32
        ;;  words (aligned to binary address xxxxx00000).  So, we do not place
        ;;  the application in the first 32-word block as we would need to
        ;;  overwrite the entire initial 32 words (including the jump to the
        ;;  bootloader).  Instead, the application's interrupt vector is
        ;;  at 0x0020 (the start of the 2nd 32-word block) and the start of
        ;;  the application proper is at 0x0024.

        ;; Here is a minimum skeleton for an application program.  Note particularly
        ;;  the 'org' statements.  Everything prior to 0x0020 will be *ignored* by
        ;;  the bootloader.  So, we set it up so the application will run the same
        ;;  whether loaded via the bootloader or burned via a high-voltage programmer.
        ;;  See also the example application programs dummy1.asm and dummy2.asm.
        ;;
        ;;      list p=16f819
        ;;      include "p16f819.inc"
        ;;      include "macros.inc"
        ;;      radix dec
        ;;      org     0x0000
        ;;         goto UserApp
        ;;      org     0x0004
        ;; ISR
        ;;         goto UserISR
        ;;      org 0x0020
        ;; UserISR
        ;;         retfie
        ;;         nop
        ;;         nop
        ;;         nop
        ;;      org 0x0024
        ;; UserApp
        ;;   ... whatever ...
        ;;      end
        ;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        ;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;; Printing a region in landscape from Emacs with enscript
        ;;    M-|
        ;;     enscript -Easm -r RET



        list p=16f819
        include "p16f819.inc"
        include "macros.inc"
        radix dec

        expand
        ;noexpand



        ;; Full paths for include files on my system are:
        ;;    /usr/share/gputils/header/
        ;;    /home/frank/pic/work/include/

;=============================================================================
```

```
;       __CONFIG 0x3F10

  __CONFIG 0x2007, _BODEN_OFF & _CP_OFF & _CPD_OFF & _PWRTE_ON & _WDT_OFF & _LVP_OFF & _MCLR_OFF &
_INTRC_IO


;;; Pin assignments

;;;; Receive into PIC from the PC (while testing use RB0 on pin 6)
;;     later use RA5/Vpp/MCLR* on pin 4

        ;; for testing, I used RB0 on pin 6
;RXPORT    equ PORTB
;RXPIN     equ 0
;RXTRIS    equ TRISB

        ;; receive serial data on RA5/Vpp/MCLR*  (pin 4)
RXPORT    equ PORTA
RXPIN     equ 5
RXTRIS    equ TRISA

KillMicroseconds macro num
        ;; x = 256 is maximum, so  5 + 3*256  =  773   i.e. (+ 5 (* 3 256))
        ;;  is the largest allowed value for num.  The smallest value
        ;;  would be x = 1 for a value for num of  5 + 3  =  8
        ;;  Accept values less than 8 but, for them, do in-line 'nop's
        if (num > 773)
            Error "KillMicroseconds called with value greater than 773"
        else
           if (num < 8)
              if (num > 0)
                 nop
                 KillMicroseconds (num - 1)
              endif
           else
              movlw ((num - 5) / 3)
              call ShortDelay
           endif
        endif
    endm



;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Start of program
;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        org     0x0000
        goto    Loader

        org     0x0004  ; And we put in a fake interrupt handler also.
ISR
        ;; This is the real interrupt vector.  It jumps to 0x0024 where
        ;;  the user must put his interrupt vector (or RTFIE)
        goto UserISR

        org 0x0008
        ;; we could start the bootloader code here since the user's code
        ;;  must skip the first 32-word block.  For now, we jump directly
        ;;  to Loader in high memory (see word address 0x0003 above).

;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        org 0x0020
UserISR
        ;; this is the beginning of the second 32-word block.  It is where the
        ;; user's code starts.  First, with 4 words allotted to the ISR
        ;; (the real ISR jumps here), then, at 0x0024, the user's startup code.
```

haha

```
        ;; User is allowed 4 instructions here to jump to actual ISR
        retfie
        nop
        nop
        nop

;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        org 0x0024
UserApp
        ;; probably, the bootloader's default user app will be to jump back to the
        ;; bootloader.
        ;Tx 'U'                  ; note we reached dummy UserApp
        goto 0                  ; start all over again

;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Loader
;;;    Note, we could put some of the loader at 0x0008, since the user's code must
;;;    not touch the first 32-word block.  For now, put all the loader in high
;;;    memory.  Note, the '818 has 1K words of program flash (1024) (0x0000-0x3ff),
;;;    and the '819 has 2K words (2048) (0x0000-0x7ff).
;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

ZFlashSize equ 0x0800            ; for 16F819
;ZFlashSize equ 0x0400            ; for 16F818
;ZFlashSize equ 0x2000            ; for 16F877

        org     0x0749  ; for 16F819


Loader
        call Initialize
        call Synchronize        ; this also gets the first command

        ;; the first command is now in W (and also in rxdata)
        ;; we are prepared only for "P" (for Program)
        IfEqLit 'P'
        goto Program

        goto UserApp
Program
        ;; We received the command "P" (for "Program").
        ;; Get the 2-byte count of how many words to program (high byte first)
        ;;  and put into the variables countH:countL used by the
        ;;  LForV / LNext loop macros.
        call [illegible]
        movwf countH
        call [illegible]
        movwf countL

        ;; ** This would be a good place to bail out if the count exceeds the
        ;; ** available memory.  Otherwise, we run the chance of overwriting
        ;; ** the bootloader code.  For now, this check is left to the (human)
        ;; ** programmer and/or the host PC downloader program.

        ;; initialize start address to 0x0020
        ;; we *always* load the user application starting at word address 0x0020,
        ;;  which is the beginning of the second 32-word block
        clrf addrH
        movlw 0x20
        movwf addrL

   LForV
        ;; get next word to be programmed into dataH:dataL (high-byte first)
        call [illegible]
        movwf dataH
        call [illegible]
        movwf dataL
```

```
        ;; If the 5 lsbits of the address are all zero, then we are about
        ;; to program the first byte of a new 32-word block, so we must
        ;; first erase that block.  PC program is responsible for delaying
        ;; a bit (at least 2 ms) after sending the word to be programmed at
        ;; the beginning of a 32-word block.
        movf addrL, W
        andlw 0x1F
        IfZ
        call EraseRow

        call FlashWrite         ; after every 4th word sent, PC should delay
                                ;   at least 2 ms for the 4-word block to
                                ;   to be burned into flash
        ;; increment the 16-bit address pointer
        incf addrL, F
        IfZ
        incf addrH, F

        movf rxdata, W          ; ** I this this instruction here is an error ** 9 August 2006 fcs
    LNext
        ;; After programming the Flash with the new user application, jump to it
        goto UserApp

SYNC equ 0xA5

;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Synchronize
;;;
;;;   (This could be named SynchronizeAndGetFirstCommand, but that's a bit verbose!)
;;;
;;;   Listen to the serial port for a while to see if the host PC wants our attention.
;;;   If it does, we will (eventually) detect a SYNC character (defined above).  Since
;;;   we are bit banging the serial port and since the PIC might wake up and start
;;;   listening to the serial port in the middle of a character, we need to make sure
;;;   we choose a SYNC character that will allow synchronization.
;;;   After at least one SYNC char, return the first non-SYNC character as the
;;;   command.
;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Synchronize
        clrf wiggles
        LFor 0x4000
          call WigglingWithDelay
          btfsc wiggles,3       ; did wiggles count up to at least 00001xxx
          goto GetSyncChar      ;  if so, jump out early
        LNext
        goto UserApp            ;  otherwise, give up and jump to the application

GetSyncChar
        ;; ok, we got a wiggle
        clrf countL             ; read up to 256 characters looking for a SYNC
SyncLoop
        call [illegible]
        sublw SYNC
        IfZ                     ; once we see at least one sync character we
        goto GetCommand         ;  are ready to look for a command
        decfsz countL, F      ; if not a sync char, have we waited long enough?
        goto SyncLoop           ; if not, keep looking
        goto UserApp            ; if so, give up and run the regular application
        ;; the problem here, is that if the serial line does not *keep* wiggling,
        ;;  we will hang forever in the call to [illegible].  I guess we could add some
        ;;  time out logic to [illegible], but ignore it for now.

GetCommand
        ;; Ok, we got at least one sync char.  Keep watching serial input until
        ;;   we get a command byte (i.e. something other than the sync char).
        ;;   If we get a "P" (for "Program") then we will burn a new application
```

"96267084"

```
        ;;   into the Flash.  If it is any other command, it is unknown, so we
        ;;   jump to the regular application.  Upon return, the command character
        ;;   is in W and also in rxdata
        ;;
        call [redacted]
        sublw SYNC
        IfZ                     ; throw away additional sync bytes
        goto GetCommand
        movf rxdata, W
        return
```

END SYNCHRONISE

```
;;; Initialize
Initialize
        Bank0
        ;; Some of this might need to be customized for the specific
        ;;  application or class of application, particularly the
        ;;  direction of I/O pins so that, e.g. application circuit
        ;;  signals are in a safe state.

        ;; We use no interrupts, and its are disabled by default, but
        ;;  make it explicit by clearing INTCON here to disable all
        ;;  interrupts.
        clrf INTCON

        ;; We do not use the timer0 interrupt.  But, if we
        ;;  did, we would need to clear the T0IF flag prior to enabling
        ;;  the interrupt (and also do this before returning from its
        ;;  int handler).
        ;bcf     INTCON, T0IF    ; turn off flag

        clrf PORTA
        clrf PORTB

        Bnk1
        ;; Set clock to 4 MHz
        movlw b'01100000'       ; 4 MHz
        movwf OSCCON

        ; turn analog/comparators off for 16LF819 and make pins digital
        movlw 0x06              ; make all PORTA pins digital
        movwf ADCON1            ;   (manual p. 82)

        ;; Port A direction
        clrf TRISA              ; all outputs (as much as possible)

        ;; Port B direction
        clrf TRISB              ; all outputs

        ;; for serial output, TXPORT,TXPIN (i.e. PORTA,1) must be output
        ;bcf TXTRIS, TXPIN       ; clear it to make RA1 an output

        ;; for serial input, RXPORT,RXPIN (i.e. at first PORTB,0 -- then later PORTA,5)
        ;;  must be an input
        bsf RXTRIS, RXPIN

        Bnk1
        ;; ** Option Register **
        movlw B'11011101'  ; prescaler middle speed (divide by 32) and no pull-ups
        movwf OPTION_REG

        Bnk0

        ;;;; init variables and queues
        clrf lastSerialIn
        clrf wiggles
```

imp.

```
;;;; Serial and timer interrupts
        ;; We are not using interrupts, but if we were, this would
        ;;   be the place to enable them.
        clrf PIR1
        clrf INTCON

        ;; Timer1, internal, divide by 8
        ;; TMR1L then increments every 8 uS with a 4 MHz crystal
        ;;  and can count up to (255 * 8) = 2040 uS ~= 2 mS
        ;; Consult TMR1L for times under ~ 2 ms and
        ;; consult TMR1H for times between about 2 mS and 1/2 second
        movlw B'00110001'
        movwf T1CON

        Bnk1
        clrf PIE1

        Bnk0
        clrf PIR1; do it again, just in case (clear interrupt flags)

        return
;;; end of 16LF819 bootloader initialize routine



;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Serial out -- inverted TTL serial in software at 9600 bps or 57600 bps
;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Tx1
;;      ;; Transmit character in W to the serial port
;;         ;; This will be a software serial port, producing fake RS232,
;;         ;;  i.e. inverted TTL serial.  So, a start bit will be high
;;         ;;  and a stop bit will be low.

;;         ;; For 9600 BPS, a bit period is  1,000,000 / 9600 --> 104-1/6 uS.
;;         ;; For 57600 BPS, a bit period is 17.36 uS.
;;         ;; For the default 4 MHz clock, an instruction takes 1 uS.  So,
;;         ;; we need to kill about 104 uS or 17 per bit.

;;         ;; Byte to be transmitted is in W, put it into txdata
;;         movwf txdata
;;         call SerialZero         ; send the logic 0 start bit (a physical high)
;;         call SerialBit          ; send each of the 8 data bits
;;         call SerialBit
;;         call SerialBit
;;         call SerialBit
;;         call SerialBit
;;         call SerialBit
;;         call SerialBit
;;         call SerialBit
;;         call SerialOne          ; send the stop bit
;;         return
;; SerialZero
;;         ;; Set TXPIN high to indicate a logic zero and kill a bit period
;;         bsf TXPORT, TXPIN
;;         goto KillBitPeriodFor9600
;; SerialOne
;;         ;; Set TXPIN low to indicate a logic one and kill a bit period
;;         nop  ; needed (btfsc instruction in SerialBit) so both paths take same time.
;;         bcf TXPORT, TXPIN
;;         goto KillBitPeriodFor9600
;; SerialBit
;;         ;; Send the current right-most bit in txdata (by shifting it into the carry)
;;         rrf txdata, f
;;         btfsc STATUS, C
;;         goto SerialOne                  ; the bit is a one
```

```
;;          goto SerialZero                 ; the bit is a zero


KillBitPeriodFor9600
        ;; So, about 10 uS have been used up so far.  kill about 92 more,
        ;; including our return instruction.
        movlw d'29'
ShortDelay
        ;; The formula (at 4 MHz) for calling ShortDelay, including the
        ;; call and also the 'movlw x' instruction is
        ;; delay in uS  =  5 + 3x
        ;; E.g.
        ;;          movlw d'29'
        ;;          call ShortDelay
        ;;  would take 5 + 3*29  = 29 uS
        movwf delayL
ShortDelayLoop
        decfsz delayL, F
        goto ShortDelayLoop
        return

;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Some macros to make the inverted logic easier to deal with
;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; IfRxZero
        ;; perform the following instruction only if the serial input is a
        ;; serial zero bit (a high voltage)
IfRxZero macro
        btfsc RXPORT,RXPIN
        endm

;;; IfRxOne
        ;; perform the following instruction only if the serial input is a
        ;; serial one bit (a low voltage)
IfRxOne macro
        btfss RXPORT,RXPIN
        endm
;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Serial in -- inverted TTL serial in software at 9600 bps or 57600 bps
;;; Use only 9600 bps to start with.
;;;
;;; We need a routine to see if the input line is wiggling.  If so, the
;;; bootloader will synchronize with the PC and accept and burn the new
;;; program, or execute the commands.  Otherwise (no wiggling), the
;;; bootloader will jump to the application program.
;;;
;;; The bootloader will check for wiggling but, later, the application
;;; program *could* also check for wiggling if it is willing to allow
;;; a reprogramming of the application.
;;;
;;; The plan is to dedicate the RA5/Vpp/MCLR* pin to serial input.  If
;;; the application program needs this pin for another purpose, then
;;; the application program would not check for it wiggling, of course.
;;;
;;; We also need a routine that will sit and wait until a character is
;;; received.
;;;
;;; The bit period at 9600 bps is 104-1/6 uS
;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Wiggling
;;  Compare the current state of the serial input line with is previous state.
;;  If it is different, the line must be wiggling!  The caller determines the
```

```
;;  time to wait between checks and how many wiggles must happen before it is
;;  taken seriously.  (The previous value meaningless on the first check, so
;;  Wiggling must be called at least twice.)  The PC is expected to send 0xAA
;;  as a synchronizing character.
;;  Caller can either check value of W returned or can check the count in
;;  wiggles.
;;  When checking at boot time for whether the serial line is wiggling, we
;;  call WigglingWithDelay but an application might prefer the quicker
;;  Wiggling (without the delay).
WigglingWithDelay
        ;call KillBitPeriodPlusFor9600
        KillMicroseconds 120
Wiggling
        movf lastSerialIn, W    ; recall last state
        IfRxOne
        goto Wone
Wzero
        bcf lastSerialIn, RXPIN ; note new value is a logic low (a one bit)
        goto Wreturn
Wone
        bsf lastSerialIn, RXPIN ; note new value is a logic high (a zero bit)
Wreturn
        xorwf lastSerialIn, W   ; W is now zero if no change

        ;;; well, I think following should be 'IfNotZ' but let's try 'IfZ'
        IfNotZ
        ;IfZ

        incf wiggles, F
        return

;;; [redacted]
        ;; Since this is inverted TTL serial, a start/zero bit is a logic
        ;; high and a stop/one bit is a logic low.  Wait for a rising edge
        ;; and take that as the beginning of a start bit.  If the ninth bit
        ;; is not a stop bit, then we must be out of sync, so we could either
        ;; abandon the byte and wait for a new start bit or we could look
        ;; backward to the most earliest zero bit after what we had thought
        ;; was a start bit, take that zero bit as a start bit and continue.
        ;; Eventually, that should get us in sync.
[redacted]
WaitForStop
        IfRxZero        ; make sure we have logic low before looking for start bit
        goto WaitForStop
WaitForStart
        IfRxOne
        goto WaitForStart
        ;; at this point, we have a rising edge that we hope is a start bit
        movlw 0x7F              ; plug in the start bit on the left
        movwf rxdata            ;  when it falls out, we've got our 8 bits
        KillMicroseconds 35     ; move to approximately the middle of the bit period
        IfRxOne                 ; if still a start bit (logic high) then
        goto WaitForStart       ;  all is well, else false alarm so go back
GetNextBit
        call KillBitPeriodFor9600
GNB2
        bcf STATUS, C           ; clear the carry flag, assuming new bit will be zero
        IfRxOne
        bsf STATUS, C           ; correct our assumption by setting the carry flag
RxShift
        rrf rxdata, F           ; shift new bit in at the left, as we shift right
        ;; if the start bit falls out, then we are done, we think
        IfC
        goto GetNextBit         ; no, a one fell out, so go back for another bit

        call KillBitPeriodFor9600 ; wait for what *should* be the stop bit
        IfRxZero
```

```
        goto GNB2               ; utoh, it wasn't a stop bit, so keep trying
                                ;   and maybe we will resynchronize eventually
        ;; ok, the stop bit occurred where expected, so we have our new character in rxdata
        ;; return it in W
        movf rxdata, W
        return

;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; End of software serial input routine, (inverted TTL serial)
;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; EraseRow
EraseRow
        ;; erase 32-word row starting at the address stored
        ;;  in addrH:addrL

        ;; set up address
        Bank2
        movf addrH, W
        movwf EEADRH            ; ms byte of address of row to erase
        movf addrL, W
        movwf EEADR             ; ls byte of address of row to erase

        ; erase the block
        Bank3
        bsf EECON1, EEPGD       ; point to program memory (not data memory)
        bsf EECON1, WREN        ; enable writes
        bsf EECON1, FREE        ; enable row erase operation
        movlw 0x55
        movwf EECON2
        movlw 0xAA
        movwf EECON2            ; magic sequence of $55 then $AA
        bsf EECON1, WR          ; start erase (CPU stalls approx 2 ms)
        nop                     ; ignored
        nop                     ; ignored

        bcf EECON1, WREN        ; disable writes
        Bank0                   ; we agree to stay in bank zero by default
        return

;;; FlashWrite
FlashWrite
        ;; Write value from dataH:dataL to address stored in addrH:addrL
        ;; This actually writes just to the buffer except when writing
        ;; the fourth word, at which point there is approximately a 2 ms
        ;; stall while the 4 words are burned to the flash.

        ;; set up the address
        Bank2
        movf addrH, W
        movwf EEADRH            ; ms byte of address of word to program
        movf addrL, W
        movwf EEADR             ; ls byte of address of word to program

        ;; set up the data
        movf dataH, W
        movwf EEDATH
        movf dataL, W
        movwf EEDATA

        ;banksel EECON1          ; i.e. Bank3
        Bank3
        bsf EECON1, EEPGD       ; point to program memory (not data memory)
        bsf EECON1, WREN        ; enable writes
        bcf EECON1, FREE        ; select flash write and not flash block erase
        movlw 0x55
        movwf EECON2
```